DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 2
Manejo de Ficheros
Parte 1

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD2 – Manejo de ficheros

ÍNDICE

1. Introducción

Si estás estudiando este módulo, es probable que ya hayas estudiado el de programación, por lo que no te serán desconocidos muchos conceptos que se tratan en este tema.

Ya sabes que cuando apagas el ordenador, los datos de la memoria RAM se pierden. Un ordenador utiliza ficheros para guardar los datos. Piensa que los datos de las canciones que oyes en mp3, las películas que ves en formato avi, o mp4, etc., están, al fin y al cabo, almacenadas en ficheros, los cuales están grabados en un soporte como son los discos duros, DVD, pendrives, etc.

A los datos que se guardan en ficheros se les llama **datos persistentes**, porque persisten más allá de la ejecución de la aplicación que los trata. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos, entre otras cosas, cómo hacer con Java las operaciones de crear, actualizar y procesar ficheros.

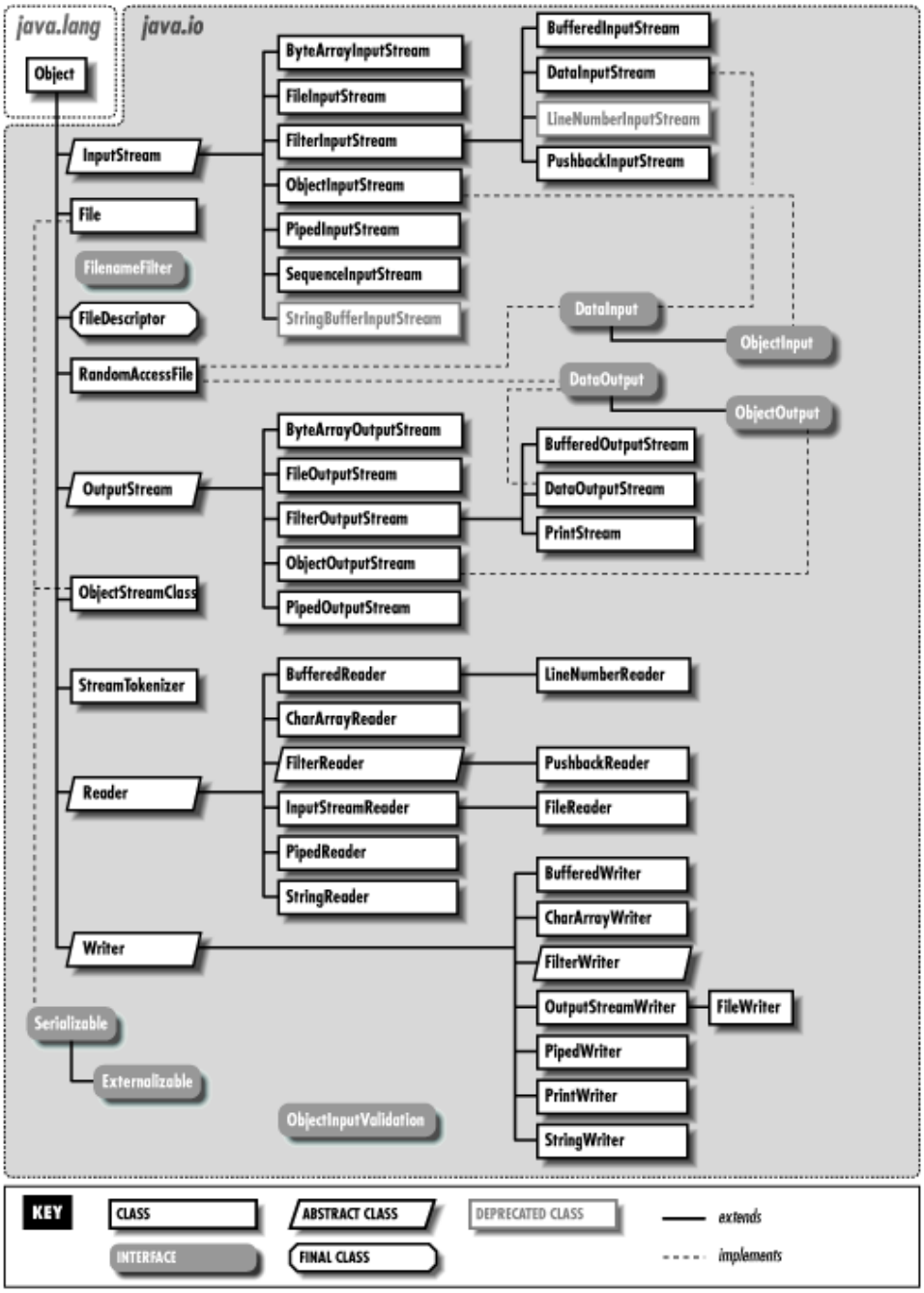
A las operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Las operaciones de E/S en Java las proporciona el paquete estándar de la API de Java denominado **java.io** que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

La librería java.io contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java. Estas clases de E/S las podemos agrupar fundamentalmente en:

- Clases para leer entradas desde un flujo de datos.
- Clases para escribir entradas a un flujo de datos.
- Clases para operar con ficheros en el sistema de ficheros local.
- Clases para gestionar la serialización de objetos.

En la siguiente imagen puedes ver las clases de las que se dispone en java.io.



Java API Reference – Web Oficial de Oracle
https://docs.oracle.com/javase/8/docs/api/

Java API Reference – JavaTPoint – I/O
https://www.javatpoint.com/java-io

ESQUEMA DE OBJETOS

Ficheros y estructuras	
File	https://docs.oracle.com/javase/8/docs/api/java/io/File.html
FilenameFilter	https://docs.oracle.com/javase/8/docs/api/java/io/FilenameFilter.html

Ficheros binarios	
FileOutputStream	https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html
DataOutputStream	https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html
FileInputStream	https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html
DataInputStream	https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html

Ficheros de texto	
FileWriter	https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html
BufferedWriter	https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html
FileReader	https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
BufferedReader	https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

Ficheros aleatorios	
RandomAccessFile	https://docs.oracle.com/javase/8/docs/api/java/io/RandomAccessFile.html

2. Clases asociadas a ficheros y directorios

Hay astantes métodos involucrados en las clases que en Java nos permiten manipular ficheros y carpetas o directorios.

Vamos a ver la clase **File** que nos permite hacer unas cuantas operaciones con ficheros, también veremos cómo filtrar ficheros, o sea, obtener aquellos con una característica determinada, como puede ser que tengan la extensión .odt, o la que nos interese, y por último, en este apartado también veremos como crear y eliminar ficheros y directorios.

2.1 Clase File

La clase **File** proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase File representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre puede ser que no exista, por esta razón habrá que controlar las posibles excepciones.

Un objeto de clase File permite examinar el nombre del archivo, descomponerlo en su rama de directorios o crear el archivo si no existe, pasando el objeto de tipo File a un constructor adecuado como **FileWriter(File f)**, que recibe como parámetro un objeto File.

Para archivos que existen, a través del objeto File, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto File, podemos hacer las siguientes operaciones con él:

- Renombrar el archivo, con el método **renameTo**(). El objeto File dejará de referirse al archivo renombrado, ya que el String con el nombre del archivo en el objeto File no cambia.
- Borrar el archivo, con el método **delete**(). También, con deleteOnExit() se borra cuando finaliza la ejecución de la máquina virtual Java.
- Crear un nuevo fichero con un nombre único. El método estático **createTempFile**() crea un fichero temporal y devuelve un objeto File que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.

- Establecer la fecha y la hora de modificación del archivo con **setLastModified**().
  Por ejemplo, se podría hacer:
  new File("prueba.txt").setLastModified(new Date().getTime());
  para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso prueba.txt.
- Crear un directorio, mediante el método **mkdir**(). También existe mkdirs(), que crea los directorios superiores si no existen.
- Listar el contenido de un directorio. Los métodos **list**() y **listFiles**() listan el contenido de un directorio. list() devuelve un vector de String con los nombres de los archivos, listFiles() devuelve un vector de objetos File.
- Listar los nombres de archivo de la raíz del sistema de archivos, mediante el método estático **listRoots**().

Mediante la clase File, podemos ver si un fichero cualquiera, digamos por ejemplo texto.txt, existe o no. Para ello, nos valemos del método exists() de la clase File. Hacemos referencia a ese fichero en concreto con el código siguiente:

```
File f = new File("C:/texto.txt");
```

En el siguiente recurso puedes ver una pequeña aplicación en la que se usa File.

B02ejer01ListarArchivos

```
String carpeta="C:/Windows";

File ruta = new File(carpeta);
if(ruta.exists()) {
      String[] listaArchivos = ruta.list();
      for(int i=0; i<listaArchivos.length; i++){
            System.out.println(listaArchivos[i]);
      }
}
```

```
New File(String)
boolean File.exists()
String[] File.list()
[].lenght
```

Pero, ¿qué pasa si nos interesa copiar un fichero, cómo lo haríamos?

Con la clase File no es suficiente, necesitamos saber más, en concreto, necesitamos hablar de los flujos, como vamos a ver más adelante.

2.2 Interface FilenameFilter.

Hemos visto como obtener la lista de ficheros de una carpeta o directorio. A veces, nos interesa ver no la lista completa, sino los archivos que encajan con un determinado criterio.

Por ejemplo, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que el que indiquemos, etc.

El interface **FilenameFilter** se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:

```
boolean accept(File dir, String nombre)
```

Este método devolverá verdadero en el caso de que el fichero cuyo nombre se indica en el parámetro nombre aparezca en la lista de los ficheros del directorio indicado por el parámetro dir.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta **c:\windows** que tengan la extensión **.exe**. Usamos try y catch para capturar las posibles excepciones, como que no exista dicha carpeta.

B02ejer02Filtrar

```
public class B0ejer02Filtrar implements FilenameFilter {
  String extension;

  B0ejer02Filtrar(String extension){
    this.extension = extension;
  }

  public boolean accept(File dir, String name){
    return name.endsWith(extension);
  }

  public static void main(String[] args) {
    try {
      File fichero=new File("c:/windows/.");
      String[] listadeArchivos = fichero.list();
      listadeArchivos = fichero.list(new B0ejer02Filtrar(".exe"));
      …
```

```
String[] File.list(FilenameFilter)
```

Clase String

http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/clases1/string.htm

2.3 Rutas de los ficheros

En algunos ejemplos que vemos en el tema, estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

```
C:\\datos\\Programacion\\fichero.txt
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo y, por tanto, persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: **File.separator** o bien el carácter `'/'`.

Podríamos hacer una función que, al pasarle una ruta, nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
static String substFileSeparator(String ruta) {
  String separador ="\\";
  try {
    if File.separator.equals(separador) ) {
      separador = "/";
    }
    return ruta.replaceAll(separador, File.separator);
  } catch (Exception e) {
    // Por si ocurre una java.util.regex.PatternSyntaxException
    return ruta.replaceAll(separador+separador, File.separator);
  }
}
```

```
File.separator
String.equals(String)
String.replaceAll(StringSearch, StringReplace)
```

2.4 Creación y eliminación de ficheros y directorios

Cuando queramos crear un fichero, podemos proceder del siguiente modo:

```
try {
  File fichero = new File("C:/prueba/miFichero.txt");
  if (fichero.createNewFile())
    System.out.println("El fichero se ha creado correctamente");
  else
    System.out.println("No ha podido ser creado el fichero");
} catch (Exception e) {
  e.getMessage();
}
```

```
File.createNewFile()
String Exception.getMessage()
```

Para borrar un fichero podemos usar la clase File, comprobando previamente si existe el fichero, del siguiente modo:

```
File fichero = new File("C:/prueba", "agenda.txt");
if (fichero.exists()) {
  fichero.delete();
}
```

```
File.exists()
File.delete()
```

Para crear directorios, podríamos hacer:

```
try {
  String directorio = "C:/prueba";
  String varios = "carpeta1/carpeta2/carpeta3";

  boolean exito = (new File(directorio)).mkdir();

  if (éxito) {
    System.out.println("Directorio: " + directorio + " creado");
    exito = (new File(varios)).mkdirs();
    if (éxito) {
      System.out.println("Directorios: " + varios + " creados");
    }
  }
} catch (Exception e){
  System.err.println("Error: " + e.getMessage());
}
```

```
File.mkdir()
File.mkdirs()
```

Para borrar un directorio con Java, tendremos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio e ir borrando con:

```
File[] ficheros = directorio.listFiles();
```

Si el elemento es un directorio, lo sabemos mediante el método isDirectory().

```
File[] File.listFiles()
boolean File.isDirectory()
```

3. Flujos

Un programa en Java, que necesita realizar una operación de entrada/salida (en adelante E/S), lo hace a través de un **flujo o stream**.

Un flujo es una abstracción de todo aquello que produce o consume información.

La vinculación de este flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

Las clases y métodos de E/S que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando. Luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red; liberando al programador de tener que saber con quién está interactuando.

Java define dos tipos de flujos en el paquete **java.io**:

- **Byte streams (8 bits)**: proporciona lo necesario para la gestión de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son **InputStream** y **OutputStream** Estas dos clases definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan read()y write() que leen y escriben bytes de datos respectivamente.
- **Character streams (16 bits)**: de manera similar a los flujos de bytes, los flujos de caracteres están determinados por dos clases abstractas, en este caso: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos read() y write() que leen y escriben caracteres de datos respectivamente.

3.1 Flujos basados en bytes (ficheros binarios)

Para el tratamiento de los flujos de bytes, hemos dicho que Java tiene dos clases abstractas que son **InputStream** y **OutputStream**.

Los archivos binarios guardan una representación de los datos en el archivo, es decir, cuando guardamos texto no guardan el texto en si, sino que guardan su representación en un código llamado UTF-8.

Las clases principales que heredan de **OutputStream**, para la escritura de ficheros binarios son:

- **FileOutputStream**: escribe bytes en un fichero. Si el archivo existe, cuando vayamos a escribir sobre él, se borrará. Por tanto, si queremos añadir los datos al final de éste, habrá que usar el constructor FileOutputStream(String filePath, boolean append), poniendo append a true.
- **ObjectOutputStream**: convierte objetos y variables en vectores de bytes que pueden ser escritos en un OutputStream.
- **DataOutputStream**, que da formato a los tipos primitivos y objetos String, convirtiéndolos en un flujo de forma que cualquier DataInputStream, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "write", como writeByte(), writefloat(), etc.

De **InputStream**, para lectura de ficheros binarios, destacamos:

- **FileInputStream**: lee bytes de un fichero.
- **ObjectInputStream**: convierte en objetos y variables los vectores de bytes leídos de un InputStream.

En el siguiente ejemplo se puede ver cómo se escribe a un archivo binario con DataOutputStream:

B02ejer03EscribirBinario

```
public class B02ejer03EscribirBinario  {
  public static void main(String args[]) {
    String texto = "Esto es un texto para almacenarlo\n
                  en el archivo binario.\n";

    String fileName =  "C:/tmp/datosbinarios.dat" ;
    try {
      FileOutputStream foStream = new FileOutputStream(fileName);
      DataOutputStream doStream = new DataOutputStream (foStream);

      doStream.writeUTF(texto);
      doStream.writeInt(5);

      doStream.close();

    } catch (IOException e) {
      System.out.print(e.getMessage());
    }
  }
}
```

```
FileOutputStream → DataOutputStream
DataOutputStream.writeUTF(String)
DataOutputStream.writeInt(int)
```

En el siguiente enlace puedes ver cómo leer de un archivo binario mediante la clase **FileInputStream**:

B02ejer03LeerBinario

```
public class B02ejer03LeerBinario  {
  public static void main(String args[]) {
    String texto;
    int entero;

    String fileName =  "C:/tmp/datosbinarios.dat" ;
    try {
      FileInputStream foStream = new FileInputStream(fileName);
      DataInputStream doStream = new DataInputStream (foStream);

      texto = doStream.readUTF();
      entero = doStream.readInt();

      doStream.close();

      System.out.println("String: " + texto);
      System.out.println("Int: " + entero);
    } catch (IOException e) {
      System.out.print(e.getMessage());
    }
  }
}
```

```
FileInputStream → DataInputStream
String DataInputStream.readUTF()
int DataInputStream.readInt()
```

3.2 Flujos basados en caracteres (ficheros de texto)

Para los flujos de caracteres, Java dispone de dos clases abstractas: **Reader** y **Writer**.

Si se usan sólo **FileInputStream**, **FileOuputStream**, **FileReader** o **FileWriter**, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro. Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando se lee o escribe, esta clase controla los accesos a disco. Así, si vamos escribiendo, se guardarán los datos hasta que haya bastantes datos como para hacer una escritura eficiente.

Al leer, la clase leerá más datos de los que se hayan pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez físicamente. Esta forma de trabajar hace los accesos a disco más eficientes y el programa se ejecuta más rápido.

En el siguiente ejemplo, puedes ver cómo se escribe en un archivo denominado **archivotexto.txt**.

B02ejer04EscribirTexto

```
      archivo = new File ("C:/tmp/archivotexto.txt");
      fw = new FileWriter (archivo);
      bw = new BufferedWriter(fw);

      bw.write("Este es un fichero de texto");
      bw.newLine();
      bw.write("que contiene varias líneas\n");
      bw.write("y se crea con BufferedWriter.\n");

      bw.flush();
      bw.close();
      fw.close();
```

```
File → FileWriter → BufferedWriter
BufferedWriter.write(String)
BufferedWriter.newLine()
BufferedWriter.flush()
BufferedWriter.close()
FileReader.close()
```

En el siguiente ejemplo, puedes ver cómo se lee un archivo guardado denominado **archivotexto.txt**.

B02ejer04LeerTexto

```
      archivo = new File ("C:/tmp/archivotexto.txt");
      fr = new FileReader(archivo);
      br = new BufferedReader(fr);

      String linea;
      while((linea=br.readLine())!=null){
            System.out.println(linea);
      }
      br.close();
      fr.close();
```

```
FileReader → BufferedReader
String BufferedReader.readLine()
BufferedReader.close()
FileReader.close()
```

4. Formas de acceso a un fichero

Hemos visto que en Java puedes utilizar dos **tipos de ficheros (de texto o binarios)** y dos **tipos de acceso a los ficheros (secuencial o aleatorio)**. Si bien, y según la literatura que consultemos, a veces se distingue una tercera forma de acceso denominada concatenación, tuberías o pipes.

- **Acceso secuencial:** En este caso los datos se leen de manera secuencial, desde el comienzo del archivo hasta el final (el cual muchas veces no se conoce a priori). Este es el caso de la lectura del teclado o la escritura en una consola de texto, no se sabe cuándo el operador terminará de escribir.
- **Acceso aleatorio:** los archivos de acceso aleatorio, al igual que lo que sucede usualmente con la memoria (RAM=Random Access Memory), permiten acceder a los datos en forma no secuencial, desordenada. Esto implica que el archivo debe estar disponible en su totalidad al momento de ser accedido, algo que no siempre es posible.
- **Concatenación (tuberías o "pipes"):** Muchas veces es útil hacer conexiones entre programas que se ejecutan simultáneamente dentro de una misma máquina, de modo que lo que uno produce se envía por un "tubo" para ser recibido por el otro, que está esperando a la salida del tubo. Las tuberías cumplen esta función.

4.1 Operaciones básicas sobre ficheros de acceso secuencial

Como operaciones más comunes en ficheros de acceso secuencial, tenemos el acceso para:

- Crear un fichero o abrirlo para grabar datos.
- Leer datos del fichero.
- Borrar información de un fichero.
- Copiar datos de un fichero a otro.
- Búsqueda de información en un fichero.
- Cerrar un fichero.

Veamos estas operaciones en unos ejemplos comentados.

En este primer ejemplo, vemos cómo se crea el fichero si no existe, o se abre para añadir datos si ya existe:

B02ejer05EscribirSecuencial

```
      String strFichero = "C:/tmp/secuencial.dat";
      int edad = 32 ;

      archivo = new DataOutputStream( new FileOutputStream(strFichero, true) );
      archivo.writeUTF( "Nombre" );
      archivo.writeUTF( "Apellidos" );
      archivo.writeInt(edad) ;
      archivo.close();
```

```
new FileOutputStream(String)        // Para borrar y escribir
new FileOutputStream(String, true)  // Para añadir
FileOutputStream → DataOutputStream
DataOutputStream.writeUTF(String)
DataOutputStream.writeInt(int)
DataOutputStream.close()
FileOutputStream.close()
```

En este ejemplo verás cómo podemos copiar un archivo origen, a otro destino, desde la línea de comandos.

B02ejer06CopiarFichero

```
      fuente = new FileInputStream(args[0]);
      destino = new FileOutputStream(args[1],true);

      int i = fuente.read();
      while (i != -1) { // mientras not EOF
            destino.write(i);
            i = fuente.read();
      }
      fuente.close();
      destino.close();
```

```
new FileInputStream(String)
int FileInputStream.read()
FileOutputStream.write(int)
FileInputStream.close()
FileOutputStream.close()
```

En este último ejemplo, has visto cómo usar la clase básica que nos permite utilizar un fichero para escritura de bytes: la clase **FileOutputStream**.

Para mejorar la eficiencia de la aplicación reduciendo el número de accesos a los dispositivos de salida en los que se almacena el fichero, se puede montar un buffer asociado al flujo de tipo **FileOutputStream**. De eso se encarga la clase **BufferedOutputStream**, que permite que la aplicación pueda escribir bytes en el flujo sin que necesariamente haya que llamar al sistema operativo para cada byte escrito.

Cuando se trabaja con ficheros de texto se recomienda usar las clases **Reader**, para entrada o lectura de caracteres, y **Writer** para salida o escritura de caracteres. Estas dos clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter **Unicode** está representado por dos bytes.

Las subclases de Writer y Reader que permiten trabajar con ficheros de texto son:

- **FileReader**, para lectura desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes.
- **FileWriter**, para escritura hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes.

También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- **BufferedWriter** se usa para montar un buffer sobre un flujo de salida de tipo **FileWriter**.
- **BufferedReader** se usa para montar un buffer sobre un flujo de entrada de tipo **FileReader**.

Operaciones básicas sobre ficheros de acceso secuencial

Veamos un ejemplo de lectura utilizando un BufferReader.

B02ejer07LeerConBuffer

```
FileReader fichero = null;
fichero = new FileReader("c:/tmp/archivotexto.txt");
BufferedReader buffer = new BufferedReader(fichero);
while ((texto = buffer.readLine()) != null) {
      System.out.println(texto);
}
fichero.close();
```

```
FileReader →  BufferedReader
new FileReader(String)
new BufferedReader(FileReader)
String BufferedReader.readLine()
FileReader.close()
```

En uno de los ejemplos anteriores, has visto como podemos grabar información a un archivo secuencial, concretamente nombre, apellidos y edad de las personas que vayamos introduciendo.

Ahora vamos a ver cómo **buscar** en un archivo secuencial, usando ese mismo ejemplo. La idea es que al ser un fichero secuencial, tenemos que abrirlo e ir leyendo hasta encontrar el dato que buscamos, si es que lo encontramos.

B02ejer08BuscarSecuencial

```
boolean seguir = true ;
String nombre = "" ;
String apellidos = "" ;
int edad = 0 ;

String busqueda = "Juan" ;
String strFic = "C:/tmp/secuencial.dat";
DataInputStream archivo = new DataInputStream( new FileInputStream(strFic) );
while (seguir) {
      nombre = archivo.readUTF();
      if (busqueda.equals(nombre)) {
            System.out.println("encontrado");
            seguir = false ;
            System.out.println("->Encontrado registro!!");
      }
      apellidos = archivo.readUTF();
      edad = archivo.readInt();
      System.out.println("Nombre: "+nombre);
      System.out.println("Apellidos: "+apellidos);
      System.out.println("Edad: "+edad);
}
archivo.close();
```

```
new FileInputStream(String)
new DataInputStream(FileInputStream)
String DataInputStream.readUTF()
Int DataInputStream.readInt()
DataInputStream.close()
```

4.2 Operaciones básicas sobre ficheros de acceso aleatorio

A menudo, no necesitas leer un fichero de principio a fin, sino simplemente acceder al fichero como si fuera una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase **RandomAccessFile** para este tipo de entrada/salida.

Esta clase:

- Permite leer y escribir sobre el fichero, no es necesario dos clases diferentes.
- Necesita que le especifiquemos el modo de acceso al construir un objeto de esta clase: sólo lectura o bien lectura y escritura.
- Posee métodos específicos de desplazamiento como **seek(long posicion)** o **skipBytes(int desplazamiento)** para poder movernos de un registro a otro del fichero, o posicionarnos directamente en una posición concreta del fichero.

Por esas características que presenta la clase, un archivo de acceso directo tiene sus registros de un tamaño fijo o predeterminado de antemano.

La clase posee dos constructores:

- **RandomAccessFile**(File file, String mode)
- **RandomAccessFile**(String name, String mode)

En el primer caso se pasa un objeto File como primer parámetro, mientras que en el segundo caso es un String. El modo es: "r" si se abre en modo lectura o "rw" si se abre en modo lectura y escritura.

A continuación puedes ver una presentación en la que se muestra cómo abrir y escribir en un fichero de acceso aleatorio.

B02ejer09EscribirAleatorio

```
RandomAccessFile miRandomFile;
String s = "Cadena a escribir\n";

miRandomFile = new RandomAccessFile( "C:/tmp/aleatorio.bin", "rw" );
miRandomFile.seek( miRandomFile.length() );
miRandomFile.writeBytes( s );
miRandomFile.close();
```

```
new RandomAccessFile( String, "rw" )
RandomAccessFile.seek( int )
RandomAccessFile.writeBytes( String )
RandomAccessFile.close()
```

También, en el segundo código descargable, se presenta el código correspondiente a la escritura y localización de registros en ficheros de acceso aleatorio.

B02ejer10OperaEscrituAlea

```
public class B02ejer10OperaEscrituAlea {
      private static final String strFichero = "C:/tmp/aleatorio.bin";
      private static final int tam_registro = 25 ;

      public static void aleatorioGraba(String s) { ... }
      public static void aleatorioNumRegistros() { ... }
      public static String aleatorioBusca(int numRegistro) { ... }

      public static void main(String args[]) {

            File fichero = new File(strFichero);
            if (fichero.exists())  fichero.delete();
            aleatorioGraba("Mi primera cadena");
            aleatorioGraba("Mi segunda cadena");
            aleatorioGraba("Mi tercera cadena");
            aleatorioGraba("Mi cuarta cadena");
            aleatorioGraba("Mi quinta cadena");

            aleatorioNumRegistros();

            System.out.println("El registro 3 contiene:");
            System.out.println(aleatorioBusca(3));
    }
}
```

```
File.delete()
new RandomAccessFile( String, "r" )
RandomAccessFile.length()
byte RandomAccessFile.readByte()
```